AWS IoT

Developer Guide

AWS IoT: Developer Guide

Table of Contents

What Is AWS IoT?

AWS Internet of Things (AWS IoT) enables secure, bi-directional communication between Internet-connected things (such as sensors, actuators, embedded devices, or smart appliances) and the AWS cloud. This enables you to collect telemetry data from multiple devices and store and analyze the data. You can also create applications that enable your users to control these devices from their phones or tablets.

AWS IoT Components

AWS IoT consists of the following components:

- **Message broker**—Provides a secure mechanism for things and IoT applications to publish and receive messages from each other. You can use the MQTT protocol to publish and subscribe. You can use the HTTP REST interface to publish.
- **Rules engine**—Provides message processing and integration with other AWS services. You can use a SQL-based language to select data from message payloads, process the data, and send the data to other services, such as Amazon S3, Amazon DynamoDB, and AWS Lambda. You can also use the message broker to republish messages to other subscribers.
- **Thing Registry**—Organizes the resources associated with each thing. You register your things and associate up to three custom attributes with each thing. You can also associate certificates and MQTT client IDs with each thing to improve your ability to manage and troubleshoot your things.
- **Thing Shadows**—Provide persistent representations of your things in the AWS cloud. You can publish updated state information to a shadow, and your thing can synchronize its state when it connects. Your things can also publish their current state to a shadow for use by applications or devices.
- **Security and identity service**—Provides shared responsibility for security in the AWS cloud. Your things must keep their credentials safe in order to send data securely to the message broker. The message broker and rules engine use AWS security features to send data securely to devices or other AWS services.

How to Get Started with AWS IoT

- To learn more about AWS IoT, see How AWS IoT Works (p. 2).
- To learn how to connect a thing to AWS IoT, see Quickstart for AWS IoT (p. 4)..

Accessing AWS IoT

AWS IoT provides the following interfaces to create and interact with your things:

- **AWS Command Line Interface (AWS CLI)**—Run commands for AWS IoT on Windows, Mac, and Linux. To get started, see the AWS Command Line Interface User Guide. For more information about the commands for AWS IoT, see iot in the *AWS Command Line Interface Reference*.
- **AWS SDKs**—Build your IoT applications using language-specific APIs. For more information, see AWS SDKs and Tools.
- **AWS IoT API**—Build your IoT applications using HTTP or HTTPS requests. For more information about the API actions for AWS IoT, see Actions in the *AWS IoT API Reference*.
- **AWS IoT Thing SDK for C**—Build IoT applications for resource-constrained things, such as microcontrollers.

Related Services

AWS IoT integrates directly with the following AWS services:

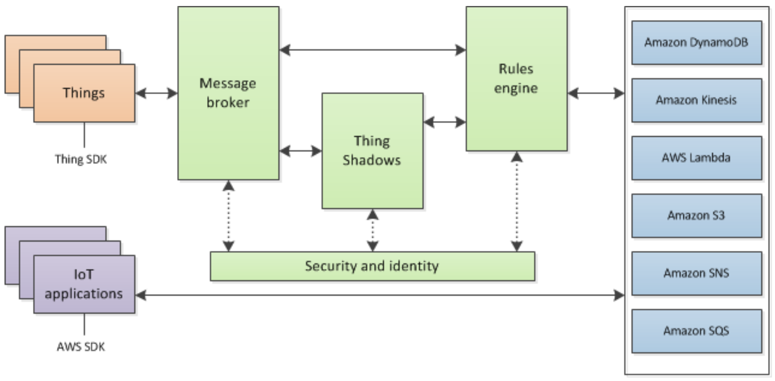
- **Amazon Simple Storage Service**—Provides scalable storage in the AWS cloud. For more information, see Amazon S3.
- **Amazon DynamoDB**—Provides managed NoSQL databases. For more information, see Amazon DynamoDB.
- **Amazon Kinesis**—Enables real-time processing of streaming data at a massive scale. For more information, see Amazon Kinesis.
- **AWS Lambda**—Runs your code on virtual servers from Amazon EC2 in response to events. For more information, see AWS Lambda.
- **Amazon Simple Notification Service**—Sends or receives notifications. For more information, see Amazon SNS.
- **Amazon Simple Queue Service**—Stores data in a queue to be retrieved by applications. For more information, see Amazon SQS.

How AWS IoT Works

AWS IoT enables Internet-connected things to connect to the AWS cloud and lets applications in the cloud interact with Internet-connected things. Common IoT applications either collect and process telemetry from devices or enable users to control a device remotely.

Things report their state by sending messages, in JSON format, to MQTT topics. Each MQTT topic has a hierarchical name, such as "myhouse/livingroom/temperature." The message broker sends each message received by a topic to all the clients subscribed to the topic.

You can create rules that define one or more actions to perform based on the data in a message. For example, you can insert, update, or query a DynamoDB table or invoke a Lambda function. Rules use expressions to filter messages. When a rule matches a message, it performs the action using the selected properties. You can use all JSON properties in a message or only the properties you need. Rules also contain an IAM role that grants AWS IoT permission to the AWS resources used to perform the action.



Each thing has a Thing Shadow that stores and retrieves state information. Each item in the state information has two entries: the state last reported by the thing and the desired state requested by an application. An application can request the current state information for a thing. The shadow responds to the request by providing a JSON document with the state information (both reported and desired), metadata, and a version number. An application can control a thing by requesting a change in its state. The shadow accepts the state change request, updates its state information, and sends a message to indicate the state information has been updated. The thing receives the message, changes its state, and then reports its new state.

Quickstart for AWS IoT

In this exercise, you will use the AWS CLI to connect a thing to AWS IoT, create rules to process messages sent by a thing, and use the Thing Registry and Thing Shadows to interact with your thing.

Tasks

Install the AWS CLI

Before you get started, you must install the latest version of the AWS CLI and configure your AWS credentials. For more information, see Getting Set Up with the AWS Command Line Interface.

To verify your installation, run the following command to list the commands available for AWS IoT:

```
aws iot help
```

The help for each subcommand describes its function, options, output, and usage. Use the following command to get help for each subcommand:

```
aws iot command help
```

For more information about formatting commands, JSON parameters, and more, see Specifying Parameter Values for the AWS Command Line Interface.

Create a Thing in the Thing Registry

To connect a thing to AWS IoT, we recommend you first create a thing in the Thing Registry. The Thing Registry allows you to keep a record of all things connected to AWS IoT. You can use the `create-thing` CLI command or the AWS IoT console to create a thing.

In a command prompt/terminal, run the following command:

```
aws iot https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta create-thing
--thing-name <thing-name>
```

This command takes a thing name, creates a new thing, and displays the thing ARN and name:

```
{
    "thingArn": "arn:aws:iot:us-east-1:<aws-account-id>:thing/lightbulb",
    "thingName": "lightbulb"
}
```

You can use the `list-things` command to confirm the thing is created in the Thing Registry:

```
aws iot https://t71u6yob51.execute-api.us-east-1.amazonaws.com/beta list-things
```

This command lists all things in the Thing Registry for your AWS account:

```
{
    "things": [
        {
            "attributes": {},
            "thingName": "lightbulb"
        }
    ]
}
```

Secure Communication Between a Thing and AWS IoT

Communication between a thing and AWS IoT is protected through the use of X.509 certificates. The process of creating and registering a certificate with AWS IoT is called *provisioning*. Certificates must be activated prior to use.

Provision a Certificate

You can provision a certificate in AWS IoT with an AWS IoT-provided public and private key pair or use your own key pair. In this example, the certificate and key pair is created by AWS IoT.

Use the create-keys-and-certificate command to create and activate a certificate :

```
aws iot create-keys-and-certificate --set-as-active
```

The output of the command contains the certificate, the public key, and the private key.

Copy the key pair from the command line and save them in separate .pem files (`privatekey.pem` and `publickey.pem`). When you copy the keys into text files, be sure to remove the embedded newlines ('\n').

Use the describe-certificate command to save the certificate to a file:

```
aws iot describe-certificate --certificate-id id --output text --query certific
ateDescription.certificatePem > cert.pem
```

`--certificate-id` specifies the part in the certificate ARN after the last slash (/): `arn:aws:iot:us-east-1:123456789012:cert/id-is-found-here`.

Now that you have created a certificate, you must create an AWS IoT policy that controls which AWS IoT operations the certificate owner can perform and attach that policy to your certificate.

Create and Attach an AWS IoT Policy to Your Certificate

Use the `create-policy` command to create an AWS IoT policy. The `--policy-document` argument contains JSON that specifies the permissions assigned to the policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action":["iot:*"],
        "Resource": ["*"]
    }]
}
```

This JSON policy document allows all AWS IoT operations on all resources. Save this text to a file and specify it in the create-policy command:

```
aws iot create-policy --policy-name PubSubToAnyTopic --policy-document
file://path-to-your-policy-document
```

Use the following attach-principal-policy command to attach the policy to your certificate:

```
aws iot attach-principal-policy --principal-arn certificate-arn --policy-name
PubSubToAnyTopic
```

Attach your Certificate to Your Thing

You use the `attach-thing-principal` CLI command to attach a certificate to a thing. The command takes two parameters:

--thing-name
: The name of the thing to which to attach the certificate.

--principal
: The ARN of your certificate.

The following shows how to call `attach-thing-principal`:

```
aws iot --endpoint attach-thing-principal --thing-name "<thing-name>" --principal
"<certificate-arn>"
```

Verify MQTT Subscribe and Publish

This section verifies you can use your certificate to communicate with AWS IoT over MQTT. You will use an MQTT client to subscribe and publish to an MQTT topic. MQTT clients require a root CA certificate to authenticate with AWS IoT. Download the root CA certificate file from root certificate.

For this walkthrough, we assume you are using Mosquitto, an open source MQTT client and broker. If you do not have Mosquitto installed, you can install it from Mosquitto.

Use the following command to retrieve your AWS account-specific AWS IoT endpoint:

```
aws iot describ-endpoint
```

This command will return an endpoint in the form of "`<random-string>.iot.us-east-1.amazonaws.com`". Use this endpoint when you use the `mosquitto_pub` and `mosquitto_sub` commands.

To subscribe to an MQTT topic, use the `mosquitto_sub` command. Provide the root CA certificate, the AWS IoT-issued certificate, and the corresponding private key:

```
mosquitto_sub --cafile path-to-cert\rootCA.pem --cert path-to-cert\cert.pem
--key path-to-cert\privateKey.pem -h your-aws-account-specific-iot-endpoint -p
8883 -q 1 -d -t topic/test -i clientid1
```

where:

`--cert` is the AWS IoT certificate.

`--key` is your private key.

`-h` is the AWS IoT service host.

`-p` is the port to use on the service host.

`-q` is the MQTT Quality of Service (QoS) level.

`-d` enables debug messages.

`-t` is the topic to publish to.

`-I` is the client ID.

> **Note**
> Ensure egress to port 8883 is allowed on your network.

The command will continue to run and display information when messages are received.

To publish a message, open a second command prompt or shell, and use the `mosquitto_pub` command:

```
mosquitto_pub --cafile certs\rootCA.pem --cert certs\cert.pem --key
certs\privateKey.pem -h data.iot.us-east-1.amazonaws.com -p 8883 -q 1 -d -t
topic/test -i clientid2 -m "Hello, World"
```

where:

`--cert` is the AWS IoT certificate.

`--key` is your private key.

`-h` is the AWS IoT service host.

`-p` is the port to use on the service host.

`-q` is the MQTT Quality of Service (QoS) level.

`-d` enables debug messages.

`-t` is the topic to publish to.

`-i` is the client ID.

`-m` is the message text to send.

Note
Ensure egress to port 8883 is allowed on your network.

This command sends a message, waits for acknowledgement, and terminates.

Configure and Test Rules

Now that you can send and receive MQTT messages through AWS IoT, you can configure rules to specify what AWS IoT should do with the messages it receives. You can configure AWS IoT rules to continuously process messages published on topics or take actions like inserting message data into a DynamoDB table or calling a Lambda function. You can configure multiple rules on a single topic.

Create an IAM Role for AWS IoT

Create an IAM role that AWS IoT can assume to perform actions when rules are triggered.

Save the following Assume Role policy document (that is, trust relationship) to a file:

```
{
"Version": "2012-10-17",
"Statement": [{
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
            "Service": "iot.amazonaws.com"
       },
      "Action": "sts:AssumeRole"
  }]
}
```

To create the IAM role, run the create-role command passing in the Assume Role policy document as follows:

```
aws iam create-role --role-name iot-actions-role --assume-role-policy-document
 file://path-to-file/trust-policy-file
```

Save the role ARN from the command output. You will need it when you create a rule.

Grant Permissions to the Role

Grant the IAM role permissions to write to DynamoDB and invoke Lambda functions. To do this, you create an IAM policy and attach the policy to your role. Save the following policy document to a file:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [ "dynamodb:*", "lambda:InvokeFunction"],
        "Resource": ["*"]
    }]
}
```

Call `create-policy` and specify the IAM policy document:

```
aws iam create-policy --policy-name iot-actions-policy --policy-document
file://IAM-policy-document-file-path
```

To attach the policy to the role, run the attach-policy-role command:

```
aws iam attach-role-policy --role-name iot-actions-role --policy-arn "policy-
ARN"
```

Create a Rule to Insert a Message into a DynamoDB Table

Use the DynamoDB console to create a DynamoDB table. The DynamoDB table must have a hash key of type String named "topic" and a range key of type Number named "timestamp." Use the defaults for all other values.

Create a rule to trigger on a topic and insert an item into the sample DynamoDB table. Rules are specified in JSON using an SQL-like syntax. The following JSON shows how to specify a rule that writes all messages sent to the topic/test topic to the sampleTable DynamoDB table:

```
{
  "sql": "SELECT * FROM 'topic/test'",
  "ruleDisabled": false,
  "actions": [{
      "dynamoDB": {
        "tableName": "HighTempSensors",
        "hashKeyField": "key",
        "hashKeyValue": "${topic(3)}",
        "rangeKeyField": "timestamp",
        "rangeKeyValud": "${timestamp()}",
        "roleArn": "arn:aws:iam::123456789012:role/iot-actions-role"
      }
    }]
}
```

Save this text to a file (for example, DynamoDbRule) and specify it in the create-topic-rule command:

```
aws iot create-topic-rule --rule-name saveToDynamoDB --topic-rule-payload
file://path-to-file/DynamoDbRule
```

Publish a message on the topic/test topic using `mosquito_pub` to invoke the rule:

```
mosquitto_pub --cafile certs\rootCA.pem --cert certs\cert.pem --key
certs\privateKey.pem -h data.iot.us-east-1.amazonaws.com -p 8883 -q 1 -d -t
topic/test -i clientid2 -m "{\"msg\" : \"Hello, World\"}"
```

Note
Make sure egress to port 8883 is allowed on your network.

To verify the data is written to the DynamoDB table, open the DynamoDB console, and open (double-click) the sampleTable to display the contents of the table.

Create a Rule to Invoke a Lambda Function

To create a Lambda function

1. Open the Lambda console, and choose **Create a Lambda Function**.
2. Choose the **hello-world** blueprint, and name the function "myHelloWorld."
3. In **Lambda function handler and role**, under **Handler**, leave the default value (index.handler). From **Role**, choose **Basic Execution Role**.
4. On the **Role** page, choose **Allow**.
5. On the new function page, choose **Next**, and then choose **Create function**.
6. On the page displayed, make a note of the function ARN. You will need it when you create a rule.

To create a rule to trigger on a topic, call the myHelloWorld Lambda function. Rules are specified in JSON using an SOL-like syntax.

The following JSON shows how to specify a rule that calls the myHelloWorld Lambda function when any message is published on the topic/test topic:

```
{
    "sql": "SELECT * FROM 'topic/test'",
    "ruleDisabled": false,
    "actions": [{
        "lambda": {
            "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my
HelloWorld"
        }
    }]
}
```

Save this text to a file (for example, LambdaRule) and specify it in the create-topic-rule command:

```
aws iot create-topic-rule --rule-name invokeLambda --topic-rule-payload
file://path-to-file/LambdaRule
```

Invoke the rule by publishing an MQTT message on the topic/test topic using the `mosquitto_pub` command:

```
mosquitto_pub --cafile rootCA.pem --cert cert.pem --key privateKey.pem -h
data.iot.us-east-1.amazonaws.com -p 8883 -q 1 -d -t topic/test -i clientid2 -m
"{\"key1\" : \"Hello, World\"}"
```

Note
Make sure egress to port 8883 is allowed on your network.

To verify the Lambda function was invoked and the message was received, go to the Lambda console and observe the CloudWatch Logs.

Using the Thing Registry and Thing Shadows

The Thing Registry allows you to keep a record of all things connected to AWS IoT. Thing Shadows allow applications to interact with the things connected to AWS IoT.

Here is the data flow:

- A thing, such as an internet-connected light bulb, is registered in the Thing Registry.
- The light bulb publishes its current state (for example, "power = on" and "color = green") to AWS IoT. AWS IoT stores the state in Thing Shadows.
- An application, such as a mobile app controlling the light bulb, uses a RESTful API to query AWS IoT for the last reported state of the light bulb.
- An application uses a RESTful API to request a change in thing state. For example, a mobile app requests that the light bulb change its color to red. The application does not have to communicate directly with the thing or be resilient to issues such as intermittent connectivity. AWS IoT will synchronize the desired state with the thing the next time the thing is connected.

For quick tour of Thing Shadows, we will use the Mosquitto client to simulate the light bulb and the AWS CLI commands to simulate the mobile application.

Register a Thing

Open a command prompt and run the create-thing command:

```
aws iot create-thing --thing-name lightbulb1
```

To confirm the thing is created in the Thing Registry, run the list-things command:

```
aws iot list-things
```

Simulate a Thing

A thing uses MQTT pub/sub (or a RESTful API) to synchronize with its shadow in AWS IoT. To report its state over MQTT, the thing publishes on topic $shadow/beta/state/`<thing-name>`. If there is an error, such as a version conflict when merging the reported state with the shadow, AWS IoT publishes an error message on topic $shadow/beta/error/`<thing-name>`. The thing should subscribe on this error topic so it will be notified of any error. To receive updates from the shadow, the thing should subscribe to topic $shadow/beta/sync/`<thing-name>`.

To use the Mosquitto client to simulate the thing, open a command prompt or shell and subscribe to the error topic with this command:

```
mosquitto_sub –cafile <rootCA-cert> --cert <thing-cert> --key <thing-private-key>
-h data.iot.us-east-1.amazonaws.com -p 8883 -q 1 -d -t
$shadow/beta/error/lightbulb1
```

Open another command prompt or shell and subscribe to the sync topic with the following command. This will simulate the receiving of updates from AWS IoT:

```
mosquitto_sub –cafile <rootCA-cert> --cert <thing-cert> --key <thing-private-key>
-h data.iot.us-east-1.amazonaws.com -p 8883 -q 1 -d -t
$shadow/beta/sync/lightbulb1
```

Open another command prompt or shell and publish a message on the state topic with the following command. This will simulate reporting the state of the thing to AWS IoT. In this example, the light bulb is reporting its current color is red:

```
mosquitto_pub –cafile <rootCA-cert> --cert <thing-cert> --key <thing-private-key>
-h data.iot.us-east-1.amazonaws.com -p 8883 -q 1 -d -t
$shadow/beta/state/lightbulb1 -m "{ \"state\": {\"reported\": { \"color\":
\"RED\" } } }"
```

Simulate an App Controlling a Thing

An application, such as a mobile app or a web application, can authenticate by using AWS credentials. It can use the AWS IoT RESTful API to get or set the state of a thing.

To get the last reported state of a thing, use the get-thing-state command:

```
aws iot-data get-thing-state --thing-name lightbulb1 output.txt && cat output.txt
```

Note
On Windows, use `&& type output.txt`.

To request an update (for example, to set state on a thing), use the update-thing-state command:

```
aws iot-data update-thing-state --thing-name lightbulb1 --payload "{ \"state\":
 {\"desired\": { \"color\": \"GREEN\" } } }"  output.txt && cat output.txt
```

This example sets the color of the light bulb to green.

Note
On Windows, use `&& type output.txt`.

Go back to the command prompt you used to subscribe the Mosquitto client to the sync topic. Verify the client received the state in a message from AWS IoT.

Delete a Thing

Use the following command to delete the thing shadow:

```
mosquitto_pub –cafile <rootCA-cert> --cert <thing-cert> --key <thing-private-key>
-h data.iot.us-east-1.amazonaws.com -p 8883 -q 1 -d -t
$shadow/beta/state/lightbulb1 -m "{ \"state\": null }"
```

Use the delete-thing command to delete the thing from the Thing Registry:

```
aws iot delete-thing --thing-name lightbulb1
```

Thing Registry for AWS IoT

AWS IoT deployments can range from a small number of mission-critical devices to large fleets with thousands of individual device units in the field. AWS IoT provides a *Thing Registry* to organize the resources associated with each device.

The Thing Registry allows you to register you units (both real devices and virtual applications) with the service, and associate up to three custom attributes with each unit. Authentication certificates used by these things can also be associated to the thing, to allow for easy viewing and diagnosis of customers' IoT fleets.

An example thing can look like this:

```
aws iot describe-thing --thing-name “MyDevice3”
```

```
{
    "thingName": " MyDevice3",
    "defaultClientId": "MyDevice3",
    "attributes": {
        "Manufacturer": "Amazon"
        "Type": "IoT Device A",
        "Serial Number": "10293847562912",
     }
}
```

A thing can then be associated with one or more certificates it can use to authenticate to the service. The registry allows for easy organization of the customer's fleet.

```
aws iot list-thing-principals --thing-name “MyDevice3”
```

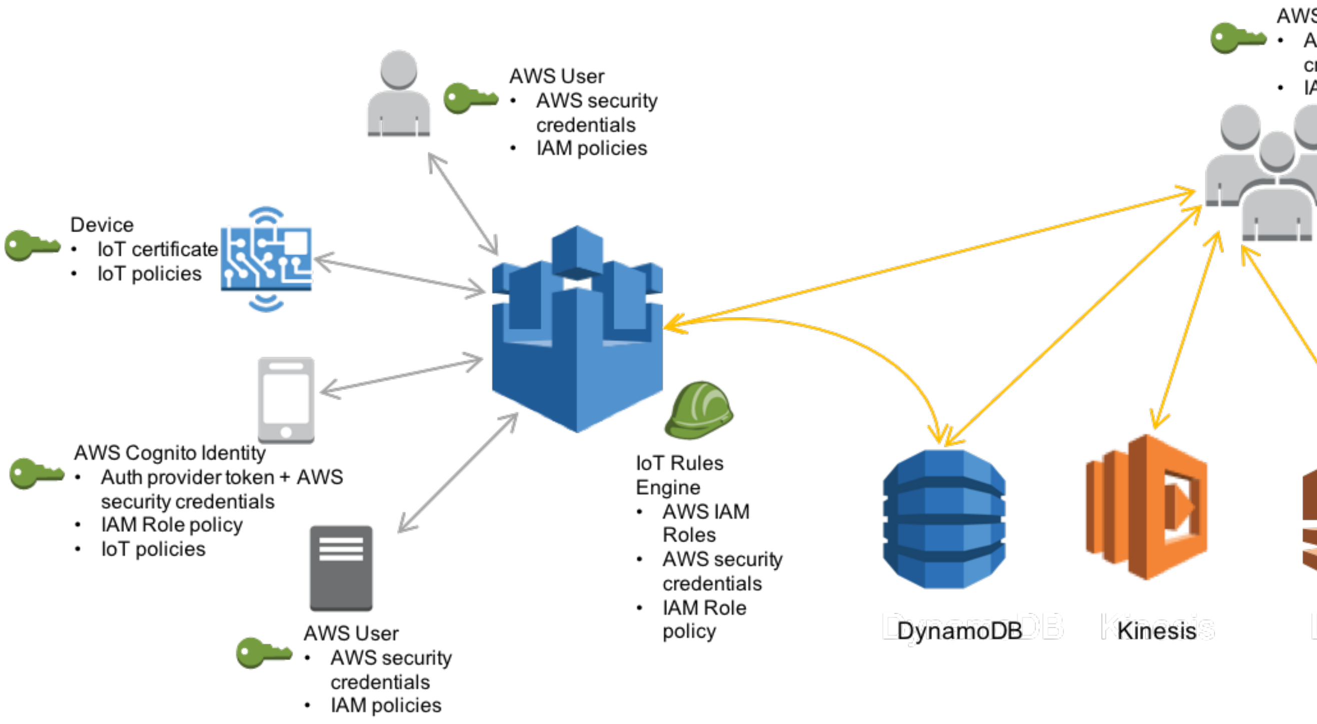
```
{
    principals: [
        "arn:aws:iot:us-east-
1:101010101010:cert/e7dc0fe48c148734641f2ba54312ba54641f2ba543fac37f2ba54348c1487ca7",

        "arn:aws:iot:us-east-
1:101010101010:cert/302ba54348c1487fe691284c5eaa6bbadc02ba54348c1487fe641fac37fb
cda5"
    ]
}
```

You can list all of your things in the Thing Registry and can search and filter based on the attributes you have provisioned. For example, you can search for all things where "manufacturer" is "Amazon." or for all things that have a serial number, and so on.

A typical use case for a device might involve using the thing name as the default MQTT client IDx. However, because IoT scenarios can involve a variety of constraints and complexities, we do not enforce a mapping between a thing’s registry name and its usage of MQTT client IDs, certificates, or shadow state. We recommend you choose a thing name for the Thing Registry and use the same name as the MQTT client ID for both the Thing Registry and the Thing Shadow service. This will add organization and convenience to your IoT fleet without taking away the flexibility of the underlying device certificate model or thing shadows.

Security and Identity for AWS IoT

The AWS IoT security model is one of shared responsibility, with an emphasis on security in the cloud. Each connected device needs a credential to access the message broker or the Thing Shadows service. All traffic to and from AWS IoT must be encrypted over TLS. Devices have the responsibility of keeping their credentials safe in order to send data securely to the message broker. After data reaches the message broker, AWS cloud security mechanisms protect data as it moves between AWS IoT and other devices or AWS services.



- You are responsible for managing credentials (X.509 certificates, AWS credentials) on your devices and policies in AWS IoT. You are responsible for assigning unique identities to each device and managing the permissions for a device or groups of devices.
- Devices connect using your choice of identity (X.509 certificates or AWS principals) over a secure connection according to AWS IoT's connection model.

- The AWS IoT message broker authenticates and authorizes all actions in your account. The message broker is responsible for authenticating your devices, securely ingesting device data, and honoring the access permissions you place on devices using policies.
- The AWS IoT rules engine forwards device data to other devices and other AWS services according to rules you define. It is responsible for leveraging AWS access management systems to securely transfer data to its final destination.

Identity in AWS IoT

AWS IoT supports three types of identity principals:

- X.509 certificates
- IAM users, groups, and roles
- Amazon Cognito identities

Each identity type enables different use cases for accessing the AWS IoT message broker and Thing Shadows service. AWS IoT provides the following guidance for selecting an identity type:

Use X.509 certificates if:

- You use MQTT as an application protocol.
- You need to support asymmetric key pairs on a device.

Use AWS users, groups, and roles if:

- You use HTTP as an application protocol.
- You want to manage identities with IAM.
- You are creating roles for AWS IoT rules and actions.

Use Amazon Cognito identities if:

- You delegate login to a Amazon Cognito-supported identity provider (such as Amazon, Facebook, Google, or an OpenID Connect compliant provider).
- You want to leverage Amazon Cognito unauthenticated identities.

X.509 Certificates

X.509 certificates provide several benefits over other identification and authentication mechanisms. X.509 certificates enable asymmetric keys to be used with devices. Your manufacturing process and your devices can be in control of keys and do not need to rely on AWS for generating security credentials. This means you can burn private keys into secure storage on a device without ever allowing the sensitive cryptographic material from leaving the device. Certificates provide stronger client authentication over other schemes such as user name/password or bearer tokens because the secret key never leaves the client.

AWS IoT authenticates certificates using the Transport Layer Security (TLS) protocol's client authentication mode. TLS is widely available in many programming languages and operating systems and is commonly used for encrypting data. In TLS client authentication, AWS IoT requests a client X.509 certificate and validates the certificate's status and AWS account against a registry of certificates. It then challenges the client for proof of the private key that corresponds to the public key contained in the certificate.

Clients must support all of the following in their TLS implementation to use AWS IoT certificates:

- TLSv1.2
- SHA-256 RSA certificate signature validation
- One of the cipher suites from the TLS cipher suite support section (add link from below)

Managing Device Certificates

You can create and manage certificates using the AWS IoT CLI. The following operations are available:

- Create a new certificate
- Activate an existing certificate
- Revoke, deactivate, or activate an existing certificate
- Transfer a certificate to another AWS account

For more details on managing device certificates, see the *AWS Command Line Interface User Guide*.

IAM Users, Groups, and Roles

IAM users, groups and roles are the standard mechanism for managing identity and authentication in AWS. You can use them to connect to AWS IoT's HTTP interfaces using the AWS SDK and CLI just like with any other AWS service.

IAM roles are also the basis for AWS IoT's security in the cloud. Roles allow AWS IoT to issue calls to other AWS resources in your account on your behalf. If you want to have a device publish its state to a DynamoDB table, for example, IAM roles allow AWS IoT to do the heavy lifting securely. For more information on IAM roles, see IAM Roles.

For message broker connections, AWS IoT authenticates IAM users, groups and roles using the SigV4 signing process. For information on authentication with AWS security credentials, see Signing AWS API Requests.

When using AWS SigV4 with AWS IoT, clients must support the following in their TLS implementation:

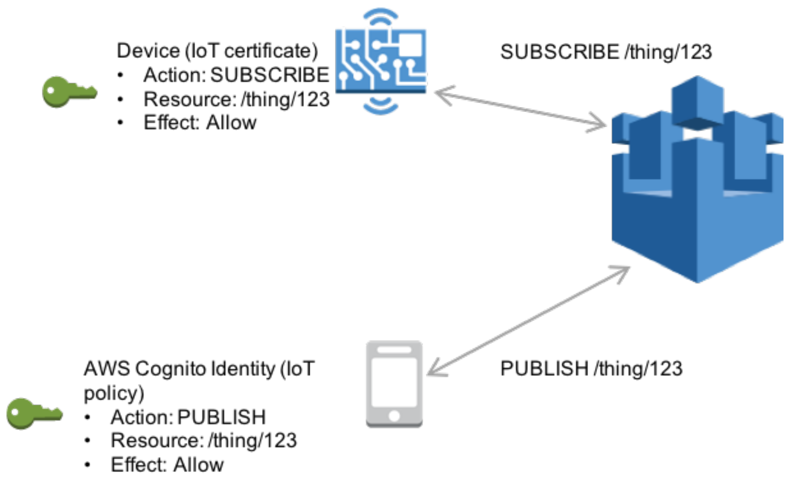
- TLSv1.2, TLSv1.1, TLSv1.0
- SHA-256 RSA certificate signature validation
- One of the cipher suites from the TLS cipher suite support section

You can manage AWS security credentials using AWS Identity and Access Management (IAM). Please see IAM for reference.

Amazon Cognito Identities

Amazon Cognito Identity allows you to use your own identity provider, or leverage other popular identity providers such as Login with Amazon, Facebook, or Google. You exchange a token from your identity provider for AWS security credentials. The credentials represent an IAM role and can be used with AWS IoT.

AWS IoT extends Amazon Cognito and allows policy attachment to Amazon Cognito Identities. You can attach a policy to a specific Amazon Cognito Identity and give fine grain permissions to an individual user of your AWS IoT application. This can be used to assign permissions between specific customers and their devices.



For more information on Amazon Cognito Identity, see Amazon Cognito Identity. For more information on using Amazon Cognito Identity policies with AWS IoT, see the AWS IoT SDK and CLI reference.

Authorization

Communication with AWS IoT follows the principal of least privilege. An identity can only execute AWS IoT operations if you grant the appropriate permission. You give permissions to identities in AWS IoT using access policies.

Policies give permissions to AWS IoT clients regardless of the authentication mechanism they use to connect to AWS IoT. You can attach AWS IoT policies to certificates or AWS Cognito Identities. You can attach IAM policies to AWS users, groups and roles.

Policy-based authorization is a powerful tool. It gives you complete control over the topics and topic filters within your AWS account. For example, consider a device connecting to AWS IoT with a certificate. You can open its access to all topics, or you can restrict its access to a single topic. The latter example allows you to assign a topic per device. For example, the device ID 123ABC can subscribe to /device/123ABC and you can grant other identities permission to subscribe to this topic, effectively opening a communication channel to this device.

AWS IoT Policies

AWS IoT policies are JSON documents and they follow the same conventions as IAM policies. For more information, see Overview of IAM Policies. An AWS IoT policy looks like the following:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action":["iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:420622145616:topic/foo/bar"]
    }]
}
```

Managing AWS IoT Policies

AWS IoT supports named policies so many identities can reference the same policy document. Named policies are versioned for rollback convenience. For more information about managing AWS IoT policies, see the AWS IoT SDK and CLI reference.

AWS IoT Policy Actions

The following actions are available for use with AWS IoT:

- iot:Publish
- iot:Subscribe
- iot:Receive
- iot:Connect
- iot:UpdateThingShadow
- iot:GetThingShadow
- iot:DeleteThingShadow

Resource Format

The following table shows the resource formats used for AWS IoT:

Action	Resource
iot:DeleteThingShadow	thing name
iot:Publish	topic ARN
iot:Subscribe	topic filter ARN
iot:UpdateThingShadow	thing name
iot:GetThingShadow	thing name

Example Policies

Certificates require a named policy, which is specified in a JSON document. These are the components of an AWS IoT policy:

Version
: Must be set to "2012-10-17".

Effect
: Must be set to "Allow" or "Deny".

Action
: Must be set to "iot":"`<operation-name>`" where <operation-name> is one of the following:

  "iot:Publish" - MQTT publish.

  "iot:Subscribe" - MQTT subscribe.

  "iot:UpdateThingShadow" - Update a thing shadow.

  "iot:GetThingShadow" - Retrieve a thing shadow.

"iot:DeleteThingShadow - Delete a thing shadow.

Resource
Must be set to an MQTT topic ARN, an MQTT topic filter ARN, or a thing name.

Topic ARN
arn:aws:iot:*<region>*:*<accountId>*:topic/*<topicName>*

Topic filter ARN
arn:aws:iot:*<region>*:*<accountId>*:topicfilter/*<topicFilter>*

Example 1

This policy allows the certificate holder to publish and subscribe to all topics in the specified AWS account.

```
{
    "Version": "2012-10-17",
        "Statement": [{
                "Effect": "Allow",
                "Action": ["iot:*"],
                "Resource": ["*"]
        }]
}
```

Example 2

This policy allows the certificate holder to publish to all topics in the specified AWS account.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": ["*"]
     }]
}
```

Example 3

This policy allows the certificate holder to publish to the topic "foo/bar" in the specified AWS account.

```
{
    "Version": "2012-10-17",
    "Statement": [{
         "Effect": "Allow",
         "Action": ["iot:Publish"],
         "Resource": ["arn:aws:iot:us-east-1:420622145616:topic/foo/bar"]
     }]
}
```

Example 4

This policy allows the certificate holder to publish to the topics "foo/bar" and "foo/baz" in the specified AWS account.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": [
                "arn:aws:iot:us-east-1:420622145616:topic/foo/bar",
                "arn:aws:iot:us-east-1:420622145616:topic/foo/baz"
        ]
    }]
}
```

Example 5

This policy prevents the certificate holder from publishing to the topic "foo/bar" in your AWS account.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Deny",
        "Action": ["iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:420622145616:topic/foo/bar"]
     }]
}
```

Example 6

This policy allows the certificate holder to subscribe to the topic filter "foo/*" in the specified AWS account.

```
{
    "Version": "2012-10-17",
    "Statement": [{
         "Effect": "Allow",
         "Action": ["iot:Subscribe"],
         "Resource": ["arn:aws:iot:us-east-1:420622145616:topicfilter/foo/*"]
     }]
}
```

Example 7

This policy allows the certificate holder to subscribe to the topic filter "foo/+/bar" in the specified AWS account.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Subscribe"],
       "Resource": ["arn:aws:iot:us-east-1:420622145616:topicfilter/foo/+/bar"]

     }]
}
```

Example 8

This policy allows the certificate holder to publish to the topic "foo" and to subscribe to the topic filter "foo/bar/*" in the specified AWS account.

```
{
    "Version": "2012-10-17",
    "Statement": [{
         "Effect": "Allow",
         "Action": ["iot:Publish"],
         "Resource": ["arn:aws:iot:us-east-1:420622145616:topic/foo"]
          },
         {
         "Effect": "Allow",
         "Action": ["iot:Subscribe"],
        "Resource": ["arn:aws:iot:us-east-1:420622145616:topicfilter/foo/bar/*"]

    }]
}
```

Example 9

This policy allows the certificate holder to publish to the topic "foo" and prevents it from publishing to the topic "bar."

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
        "Resource": ["arn:aws:iot:us-east-1:420622145616:topic/foo"]
         },
         {
         "Effect": "Deny",
         "Action": ["iot:Publish"],
         "Resource": ["arn:aws:iot:us-east-1:420622145616:topic/bar"]
    }]
}
```

Example 10

This policy allows the certificate holder to subscribe to the topic filter "foo/bar."

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Subscribe"],
        "Resource": ["arn:aws:iot:us-east-1:420622145616:topicfilter/foo/bar"]
    }]
}
```

Example 11

This policy allows the certificate holder to delete all thing shadows in the specified AWS account.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:DeleteThingShadow"],
        "Resource": ["*"]
     }]
}
```

Transport Security

The AWS IoT’s message broker and Thing Shadows service encrypt all communication with TLS. TLS is used to ensure the confidentiality of the application protocols (MQTT, HTTP) AWS IoT supports. TLS is readily available in a variety of programming languages and operating systems.

For MQTT, TLS encrypts the connection between the device and the broker. TLS client authentication is used to identity devices to AWS IoT. For HTTP, TLS encrypts the connection between the device and the broker. Authentication is delegated to AWS signature version 4.

TLS Cipher Suite Support

AWS IoT supports the following cipher suites:

- ECDHE-ECDSA-AES128-GCM-SHA256 (recommended)
- ECDHE-RSA-AES128-GCM-SHA256 (recommended)
- ECDHE-ECDSA-AES128-SHA256
- ECDHE-RSA-AES128-SHA256
- ECDHE-ECDSA-AES128-SHA
- ECDHE-RSA-AES128-SHA
- ECDHE-ECDSA-AES256-GCM-SHA384
- ECDHE-RSA-AES256-GCM-SHA384
- ECDHE-ECDSA-AES256-SHA384
- ECDHE-RSA-AES256-SHA384
- ECDHE-RSA-AES256-SHA
- ECDHE-ECDSA-AES256-SHA
- AES128-GCM-SHA256
- AES128-SHA256
- AES128-SHA
- AES256-GCM-SHA384
- AES256-SHA256
- AES256-SHA

Service Limits for Security and Identity

There are no practical limits to the number of certificates and policies you can create with AWS IoT.

- You can attach up to 10 policies to an AWS IoT certificate.
- You can keep up to 5 versions of a named policy.
- Policy document size is limited to 2048 characters (excluding white space).

Message Broker for AWS IoT

The AWS IoT message broker is a pub/sub broker service that enables sending and receiving messages to and from AWS IoT. When communicating with AWS IoT, a client sends a message addressed to a topic like "Sensor/temp/room1." The message broker, in turn, sends the message to all clients that have registered to receive messages for that topic. The act of sending the message is referred to as publishing. The act of registering to receive messages for a topic filter is referred to as subscribing.

The topic namespace is isolated for each AWS account and region pair. For example, the "Sensor/temp/room1" topic for an AWS account is independent from the "Sensor/temp/room1" topic for another AWS account. This is true of regions, too. The "Sensor/temp/room1" topic in the same AWS account in us-east-1 is independent from the same topic in us-west-2. AWS IoT does not support sending and receiving messages across AWS accounts and regions.

The message broker maintains a list of all client sessions and the subscriptions for each session. When a client connects to the broker, it can choose to start a *clean session* or ask the broker to keep a persistent session state across connections and disconnections. When a message is published on a topic, the broker checksfor sessions with subscriptions that map to the topic. The broker then forwards the publish message to all sessions that have a currently connected client. For all matching sessions that do not have a connected client, the broker saves the message for delivery when they connect.

Protocols

The message broker supports the use of the Message Queue Telemetry Transport (MQTT) protocol to publish and subscribe. It supports the use the of the HTTPS protocol to publish.

MQTT

MQTT is a widely adopted lightweight messaging protocol designed for constrained devices. For more information about MQTT, go to MQTT.org. Although the AWS IoT message broker implementation is based on MQTT v3.1.1, it deviates from the specification as described in the implementation notes.

In AWS IoT, subscribing to a topic with Quality of Service (QoS) 0 means a message will be delivered zero or more times. A message may be delivered more than once. Messages delivered more than once may be sent with a different packet ID. In these cases, the DUP flag is not set.

AWS IoT does not support publishing and subscribing with QoS 2. The AWS IoT message broker does not send a PUBACK or SUBACK when QoS 2 is requested.

When responding to a connection request, the broker sends a CONNACK message. This message contains a flag indicating if the connection is resuming a previous session. The value of this flag may be incorrect when two MQTT clients connect with the same client ID simultaneously.

When a client subscribes to a topic, there may be a time delay between the time the broker sends a SUBACK and the time the client starts receiving new matching messages.

The MQTT specification provides a provision for the publisher to request that the broker retain the last message sent to a topic and send it to all future topic subscribers. AWS IoT does not support retained messages. If a request is made to retain messages, the connection is disconnected.

The message broker uses the client ID to identify each client . The client ID is passed in from the client to the broker as part of the MQTT payload. Two clients with the same client ID are not allowed to be concurrently connected to the message broker. When a client connects to the broker using a client ID that another client is using, a CONNACK will be sent to both clients and the existing client will be disconnected.

The message broker does not supports persistent sessions (clean session set to 0). All sessions are assumed to be clean sessions and messages are not stored across sessions.

HTTP

The message broker supports clients connecting with the HTTP protocol using a REST API. Clients can publish by sending a POST message to `<AWS IoT Endpoint>/topics/<url_encoded_topic_name>?qos=1"`.

Topics

The message broker uses topics to route messages from publishing clients to subscribing clients. The forward slash '/' is used to separate topic hierarchy. The following tables lists the wildcards that can be used in the topic filter when subscribing.

Topic Wild-cards

Wildcard	Description
#	AWS IoT supports the use of '#' as a wildcard when subscribing to a topic. The '#' character must be the last character in the topic to which you are subscribing. The '#' works as a wildcard by matching the current tree and all subtrees. For example, a subscription to "Sensors/#" will receive messages published to "Sensor/," "Sensor/temp," "Sensor/temp/room1," but not the messages published to "Sensor."
+	AWS IoT supports the use of '+' as a wildcard when subscribing to a topic. The '+' character matches exactly one item in the topic hierarchy. For example, a subscription to "Sensors/+/room1" will receive messages published to "Sensor/temp/room1," "Sensor/moisture/room1," and so on.

Reserved Topics

Any topics beginning with '$' are considered reserved and are not supported for publishing and subscribing except when working with Thing Shadows. For more information, see Thing Shadows.

Message Broker Limits

The following table describes limits in AWS IoT. Each limit applies on a per-region basis.

Topic Length Limit (256 bytes UTF-8)	The topic passed to the message broker in publish messages can not exceed 256 bytes UTF-8
Maximum Number of Slashes in Topic & Topic Filter (8)	A topic provided while publishing a message or a topic filter provided while subscribing can at most have 8 slashes('/').
Message Size Limit (128KB)	The payload for every publish message is limited to 128KB. An attempt to publish a message above this size will be rejected by the AWS IoT service.
Throughput per connection (512 KB/s)	AWS IoT limits the ingress and egress rate on each client connection to 512 KB/s. If a client attempts to send or receive data at a rate higher than this it will be throttled to this throughput.
Client ID Size Limit (128 bytes UTF-8)	The client ID size is limited to 128 bytes encoded in UTF-8.
Restricted Client ID prefix (GEN/)	Client IDs can not begin with 'GEN/' as it is reserved for internally generated Client IDs.
Max Subscriptions per Subscribe call (8)	A single subscribe call is limited to request a maximum of 8 subscriptions.
Subscriptions per Session (50)	The message broker limits each client session to subscribe to up to 50 subscriptions. A subscribe request that pushes the total number of subscriptions past 50 will result in the connection being disconnected.
Message retention time for offline clients (1 day)	The message broker supports persistent sessions (clean session set to 0) and stores the messages for offline clients for up to one day from the time the message was published.

Rules for AWS IoT

Rules give your things the ability to interact with a variety of AWS services. Rules are analyzed and actions are performed based on the MQTT topic stream. You can use rules to support tasks like these:

- Build a time series data store using Amazon DynamoDB.
- Save a firmware file to Amazon S3.
- Send a push notification to all users using Amazon SNS.
- Publish to an Amazon SQS queue.
- Invoke a Lambda function to extract data.
- Process messages from a large number of devices using Amazon Kinesis.

Before AWS IoT can perform these actions, you must grant it permission to access your AWS resources on your behalf. When the actions are performed, you incur the standard charges for the AWS services that you use.

Contents

Granting AWS IoT the Required Access

You use IAM roles to control the AWS resources to which each rule has access. Before you create a rule, you must create an IAM role with a policy that allows access to the required AWS resources. AWS IoT assumes this role when executing a rule.

To create an IAM role (AWS CLI)

1. Use the create-role command to create an IAM role:

```
aws iam create-role --role-name my-iot-role --assume-role-policy-document
file://my-iot-role-trust-policy-document.json
```

Specify the following trust policy document, which grants AWS IoT permission to assume the role:

```
{
    "Version":"2012-10-17",
    "Statement":[{
        "Effect": "Allow",
        "Principal": {
            "Service": "iot.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
    }]
}
```

Note the Amazon Resource Name (ARN) of the role in the command output.

```
{
  "Role": {
      "AssumeRolePolicyDocument": "url-encoded-json",
      "RoleId": "AKIAIOSFODNN7EXAMPLE",
      "CreateDate": "2015-09-30T18:43:32.821Z",
      "RoleName": "my-iot-role",
      "Path": "/",
      "Arn": "arn:aws:iam::123456789012:role/my-iot-role"
  }
}
```

2. Use the create-policy command to grant AWS IoT access to your AWS resources upon assuming the role:

```
aws iam create-policy --policy-name my-iot-policy --policy-document file://my-
iot-policy-document.json
```

The following is an example policy document, which grants AWS IoT administrator access to DynamoDB:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": "dynamodb:*",
        "Resource": "*"
    }]
}
```

Note the ARN of the policy in the command output.

```
{
    "Policy": {
        "PolicyName": "my-iot-policy",
        "CreateDate": "2015-09-30T19:31:18.620Z",
        "AttachmentCount": 0,
        "IsAttachable": true,
        "PolicyId": "ZXR6A36LTYANPAI7NJ5UV",
        "DefaultVersionId": "v1",
        "Path": "/",
        "Arn": "arn:aws:iam::123456789012:policy/my-iot-policy",
        "UpdateDate": "2015-09-30T19:31:18.620Z"
    }
}
```

3. Use the attach-role-policy command to attach your policy to your role:

```
aws iam attach-role-policy --role-name my-iot-role --policy-arn
"arn:aws:iam::123456789012:policy/my-iot-policy"
```

Creating a Rule

You configure rules to route data from your connected things. Rules consist of the following:

Rule name
: The name of the rule.

Optional description
: The purpose of the rule.

SQL statement
: A simplified SQL syntax to filter messages received on an MQTT topic and push the data elsewhere. For more information, see AWS IoT SQL Reference (p. 32).

One or more actions
: The actions AWS IoT takes when executing the rule. For example, you can insert data in a DynamoDB table, write data to an Amazon S3 bucket, publish to an Amazon SNS topic, or invoke a Lambda function.

When you create a rule, be aware of how much data you are publishing on topics. If you create rules that include a wildcard topic pattern, they might match a large percentage of your messages, and you might need to increase the capacity of the AWS resources used by the target actions. Also, if you create a republish rule that includes a wildcard topic pattern, you can end up with a circular rule that causes an infinite loop.

To create a rule (AWS CLI)

Use the create-topic-rule command to create a rule:

```
aws iot create-topic-rule --rule-name my-rule --topic-rule-payload file://my-
rule.json
```

The following is an example payload file with a rule that inserts all messages sent to the iot/test topic into the specified DynamoDB table. The SQL statement filters the messages, and the role ARN grants AWS IoT permission to write to the DynamoDB table.

```
{
  "sql": "SELECT * FROM 'iot/test'",
  "ruleDisabled": false,
  "actions": [{
      "dynamoDB": {
          "tableName": "my-dynamodb-table",
          "roleArn": "arn:aws:iam::123456789012:role/my-iot-role",
          "hashKeyField": "key",
          "hashKeyValue": "${topic(3)}",
          "rangeKeyField": "timestamp",
          "rangeKeyValue": "${timestamp()}"
      }
  }]
}
```

The following is an example payload file with a rule that invokes a Lambda function:

```
{
  "sql": "expression",
  "ruleDisabled": false,
  "actions": [{
      "lambda": {
          "functionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
lambda-function"
      }
  }]
}
```

The following is an example payload file with a rule that publishes to an Amazon SNS topic:

```
{
  "sql": "expression",
  "ruleDisabled": false,
  "actions": [{
      "sns": {
          "targetArn": "arn:aws:sns:us-west-2:123456789012:my-sns-topic",
          "roleArn": "arn:aws:iam::123456789012:role/my-iot-role"
      }
  }]
}
```

The following is an example payload file with a rule that republishes on a different MQTT topic:

```
{
  "sql": "expression",
  "ruleDisabled": false,
  "actions": [{
      "republish": {
          "topic": "my-mqtt-topic",
          "roleArn": "arn:aws:iam::123456789012:role/my-iot-role"
      }
  }]
}
```

Viewing Your Rules

Use the list-topic-rules command to list your rules:

```
aws iot list-topic-rules
```

Use the get-topic-rule command to get information about a rule:

```
aws iot get-topic-rule --rule-name my-rule
```

Troubleshooting a Rule

If you are having an issue with your rules, you should enable CloudWatch Logs. By analyzing your logs, you can determine whether there are authorization issues or whether a WHERE clause condition did not match. For more information, see Troubleshooting AWS IoT (p. 67).

Deleting a Rule

When you are finished with a rule, you can delete it.

To delete a rule (AWS CLI)

Use the delete-topic-rule command to delete a rule:

```
aws iot delete-topic-rule --rule-name my-rule
```

AWS IoT SQL Reference

This reference focuses on the differences between ANSI SQL and AWS IoT SQL. If you are not familiar with ANSI SQL already, see the W3Schools SQL Tutorial.

SELECT Statements

All rules include a SQL statement that consists of SELECT with a topic filter and an optional WHERE clause filter. By using the same topic semantics as an MQTT subscriber, any data published on a topic that matches this filter would be subscribed by the rule.

All data processed by a query is assumed to be JSON. A flat JSON document should require SQL that is identical to traditional SQL. If the JSON is nested, see JSON Extensions (p. 36).

As with ANSI SQL, white space is insignificant and keywords are not case-sensitive. Strings and JSON properties remain case-sensitive. In our examples, we capitalize all keywords, following common practice for SQL.

FROM Clause

In ANSI SQL, you select data from tables. In AWS IoT SQL, you select data from JSON properties in messages to MQTT topics.

The data source is written as a function, as shown in this example:

```
SELECT * FROM mqtt('com.example/sensors/+')
```

However, the default is MQTT, so you can simply use a string, as shown in this example:

```
SELECT * FROM 'com.example/sensors'
```

WHERE Clause

The WHERE clause filters messages based on criteria for the JSON properties. For example, suppose you have a topic filter from an MQTT topic, `iot/thing/#`. The following is an example JSON payload that could be published by a device:

```
{
    "deviceid" : "iot123",
    "temp" : 54.98,
    "humidity" : 32.43,
    "coords" : {
        "latitude" : 47.615694,
        "longitude" : -122.3359976
    }
}
```

You could use the following SQL statement in your rule to query the `iot/thing/#` topic and extract the sensor data when the `temp` field is above 50.

```
SELECT * FROM 'iot/thing/#' WHERE temp > 50
```

You can use most of the expressions allowed in ANSI SQL. The following expressions are allowed in AWS IoT.

Token	Meaning	Example
=	Equal, comparison	color = 'red'
<>	Not equal, comparison	color <> 'red'
AND	Logical AND	color = 'red' AND siren = 'on'
OR	Logical OR	color = 'red' OR siren = 'on'
()	Parenthesis, grouping	color = 'red' AND (siren = 'on' OR isTest)
+	Addition, arithmetic	4 + 5

Token	Meaning	Example
-	Subtraction, arithmetic	5 - 4
/	Division, arithmetic	20 / 4
*	Multiplication, arithmetic	5 * 4
%	Modulo division, arithmetic	20 % 6
<	Less than, comparison	5 < 6
<=	Less than or equal, comparison	5 <= 6
>	Greater than, comparison	6 > 5
>=	Greater than or equal, comparison	6 >= 5
CASE ... WHEN ... THEN ... ELSE ... END	Case statement	CASE location WHEN 'home' THEN 'off' WHEN 'work' THEN 'on' ELSE 'silent' END

Functions

You can use the following built-in functions in the SELECT or WHERE clauses of your SQL expressions.

Function	Description
abs(*number*)	Returns the absolute value.
accountId()	Returns the account ID of the MQTT client sending the message, or undefined if the message didn't come through MQTT.
asin(*number*)	Returns the arcsine.
atan(*number*)	Returns the arctangent.
bitand(*number1*, *number2*)	Returns the result of a bitwise AND operation.
ceil(*number*)	Returns the result of rounding up to the nearest integer.
chr(*number*)	Returns the ASCII character represented by *number*.
clientId()	Returns the client ID of the MQTT client sending the message, or undefined if the message didn't come through MQTT.
concat(*string1*, *string2*)	Returns the concatenation of two strings.

Function	Description
cos(*number*)	Returns the cosine.
cosh(*number*)	Returns the hyperbolic cosine.
endswith(*input*, *suffix*)	Returns true if *input* ends with *suffix*.
exp(*number*)	Returns e to the power of the specified number.
floor(*number*)	Returns the result of rounding down to the nearest integer.
ln(*number*)	Returns the natural logarithm.
log(*n*, *m*)	Returns the logarithm of *n* base *m*.
lower(*string*)	Returns the result of converting all characters to lowercase.
lpad(*string*, *n*)	Adds *n* spaces to the left side of *string*.
ltrim(*string*)	Removes all white space from the left side of *string*.
md2(*string*)	Returns the MD2 hash value.
md5(*string*)	Returns the MD5 hash value.
mod(*m*, *n*)	Returns the remainder of *m* divided by *n*.
nanvl(*value*, *default*)	Returns *value* if it's non-null, and *default* otherwise.
power(*m*, *n*)	Returns *m* raised to the *n*th power.
regexp_matches(*input*, *pattern*)	Returns true if regular expression *pattern* is matched by *input*.
regexp_replace(*source*, *pattern*, *replacement*)	Performs a regular expression replacement on *source*.
regexp_substr(*source*, *pattern*)	Returns the first substring of *source* matched by *pattern*.
remainder(*m*, *n*)	Returns the remainder of *m* divided by *n*.
replace(*source*, *substring*, *replacement*)	Returns *source* with all occurrences of *substring* replaced by *replacement*.
round(*number*, *precision*)	Returns the result of rounding *number* to *precision* decimal places. If *precision* is 0, the function rounds to the nearest whole number.
rpad(*string*, *n*)	Adds *n* spaces to the right side of *string*.
rtrim(*string*)	Removes all white space from the right side of *string*.
sign(*number*)	Returns a value indicating the sign of a number. If number < 0, then -1. Else, if number = 0, then 0. Else, if number > 0, then 1.
sin(*number*)	Returns the sine.

Function	Description
sinh(*number*)	Returns the hyperbolic sine.
sqrt(*number*)	Returns the square root.
startswith(*input*, *prefix*)	Returns true if *input* starts with *prefix*.
tan(*number*)	Returns the tangent.
tanh(*number*)	Returns the hyperbolic tangent.
traceId()	Returns the trace ID of the MQTT message, or undefined if the message didn't come through MQTT.
trunc(*number*, *precision*)	Returns the result of truncating *number* to *precision* decimal places.
upper(*string*)	Returns the result of converting all characters to uppercase.
sha1(*string*)	Returns the SHA-1 hash value.
sha224(*string*)	Returns the SHA-224 hash value.
sha256(*string*)	Returns the SHA-256 hash value.
sha512(*string*)	Returns the SHA-512 hash value.
rand()	Returns a random number between 0 and 1.
newuuid()	Returns a random 20-byte UUID.
timestamp()	Returns the current Unix timestamp, as observed by the current server.

JSON Extensions

You can use the following extensions to ANSI SQL syntax to make it easier to work with nested JSON objects.

"." Operator

This operator functions identically to ANSI SQL and JavaScript.

".." Operator

This operator accesses members in inner objects. Unlike the "." operator, it searches arbitrarily deep to find a match.

For example, suppose you have the following query:

```
SELECT * FROM 'topic'
WHERE a..b = 3
```

The following three JSON examples are all matches:

```
{"a": {"b":3}}
```

```
{
    "a": {
        "x": {
            "y": {"b": 3}
        }
    }
}
```

```
{
    {"a": [
        {"b": 3},
        {"b": 4}
    ]}
}
```

Double dots ".." are used inside the WHERE clause to mean "if any matching value satisfies this Boolean expression." For example, in the third of the preceding examples, the expression "a..b" matches two values (3 and 4). The full expression "a..b = 3" is evaluated as "result1 = 3 OR result2 = 3". When used inside the SELECT clause as a single value, only the last match is used. Again, in the third of the preceding examples, if the SQL were rewritten as:

```
SELECT a..b
FROM 'topic'
```

It would produce this JSON result:

```
{"b": 3}
```

"*" Operator

This functions exactly like the * wildcard in ANSI SQL. It's used in the SELECT clause only and creates a new JSON object.

Applying a Function to an Attribute Value

The following is an example JSON payload that could be published by a device:

```
{
    "deviceid" : "iot123",
    "temp" : 54.98,
    "humidity" : 32.43,
    "coords" : {
        "latitude" : 47.615694,
        "longitude" : -122.3359976
    }
}
```

The following example applies a function to an attribute value in a JSON payload:

```
SELECT temp, md5(deviceid) AS hashed_id FROM topic/#
```

The result of this query is the following JSON object:

```
{
   "temp": 54.98,
   "hashed_id": "e37f81fb397e595c4aeb5645b8cbbbd1"
}
```

Substitution Templates

You can use a substitution template to provide data when AWS IoT performs an action. The syntax is `${`*expression*`}`, where *expression* can be any expression that's valid in a WHERE or SELECT clause (for example, `${jsonProperty1}-${topic()}`).

Thing Shadows for AWS IoT

A *thing shadow* is a JSON document that is used to store and retrieve current state information for a thing (device, app, and so on). The Thing Shadows component maintains a thing shadow for each thing you connect to AWS IoT. You can use thing shadows to get and set the state of a thing over MQTT or HTTP, regardless of whether the thing is connected to the Internet.

Each thing shadow is uniquely identified by its name.

Contents

Thing Shadow Documents

Thing Shadows supports full JSON so you can store values, objects, and arrays in the shadow document.

Contents

Document Structure

A thing shadow document has the following structure:

state

desired
: The desired state of the thing. Applications can write to this portion of the document to update the state of a thing without having to directly connect to a thing.

reported
: The reported state of the thing. Things write to this portion of the document to report their new state. Applications read this portion of the document to determine the state of a thing.

metadata
: Information about the data stored in the `state` section of the document. This includes timestamps, in epoc time, for each attribute in the `state` section, which allows you to determine when the state was updated.

version
: The document version. Every time the document is updated, this version number is incremented. This can be used to ensure the version of the document being updated is the most recent.

clientToken
: A string unique to the device that allows you to associate responses with requests in an MQTT environment.

timestamp
: The global timestamp of a document records when the message was transmitted by the AWS IoT service. By using the timestamp in the message and metadata information for individual attributes in the desired or reported section, a thing can determine how old an updated item is, even if it doesn't feature an internal clock.

Thing Shadow Versions

Thing shadows support versioning on every update message (both request and response), which means that with every update of a thing shadow, the version of the JSON document is incremented. This ensures two things:

- A client can receive an error if it attempts to overwrite a shadow using an older version number. The client is informed it needs to resync before it can update a thing shadow.
- A client can decide not to act on a received message if the message has a lower version than the version stored by the client.

In some cases, a client might not care whether an update is the latest version. It can bypass version matching by not submitting a version.

Client Token

You can use a client token with MQTT-based messaging to verify a request and the request response contain the same client token. This ensures the response and request are associated.

Example Document

Here is an example thing shadow document:

```
{
    "state" : {
        "desired" : {
          "color" : "RED",
          "sequence" : [ "RED", "GREEN", "BLUE" ]
        },
        "reported" : {
          "color" : "GREEN"
        }
    },
    "metadata" : {
        "desired" : {
            "color" : {
                "timestamp" : 12345
            },
            "sequence" : {
                "timestamp" : 12345
            }
        },
        "reported" : {
            "color" : {
                "timestamp" : 12345
            }
        }
    },
    "version" : 10,
    "clientToken" : "UniqueClientToken",
    "timestamp": 123456789
}
```

Empty Fields

Empty desired and reported fields are removed from a thing shadow document automatically. A document only contains a desired node if it has state in that node. For example, the following is a valid state document with no desired node:

```
{
    "reported" : { "temp": 55 }
}
```

The reported node can also be empty:

```
{
    "desired" : { "color" : "RED" }
}
```

Finally, if an update caused the desired or reported field to become empty, it is removed from the document. It is possible that a thing shadow document will not contain desired and reported fields.. In that case, the shadow document is empty. For example, this is a valid document:

```
{
}
```

Arrays

Thing shadows support arrays, but treat them as normal values in that an update to an array replaces the whole array. It is not possible to update part of an array.

Initial state:

```
{
    "desired" : { "colors" : ["RED", "GREEN", "BLUE" ] }
}
```

Update:

```
 {
    "desired" : { "colors" : ["RED"] }
}
```

Final state:

```
{
    "desired" : { "colors" : ["RED"] }
}
```

Arrays can't have null values. For example, the following array is not valid and will be rejected.

```
{
    "desired" : {
        "colors" : [ null, "RED", "GREEN" ]
    }
}
```

Limits

The following limits apply for the use of thing shadows:

- The maximum size of a state document in JSON cannot exceed 8 KB.
- You can store an unlimited number of JSON objects per account.
- A thing shadow is deleted by AWS IoT if it has not been updated or retrieved in more than 12 months.

Using Thing Shadows

AWS IoT provides three methods for working with thing shadows:

UPDATE
: Creates a thing shadow if it doesn't exist, or updates the content of a thing shadow with the data provided in the request. The data is stored with timestamp information to indicate when it was last updated. Messages are sent to all subscribers with the difference between reported and desired state (delta). Things or apps that receive a message can perform an action based on the difference between reported and desired states. For example, a device can update its state to the desired state, or an app can update its UI to show the change in the device's state.

GET
: Retrieves the latest state stored in the thing shadow (for example, during startup of a device to retrieve configuration and the last state of operation). This method returns the full JSON document, including metadata.

DELETE
: Deletes a thing shadow, including all of its content. This removes the JSON document from the data store. You can't restore a thing shadow you deleted, but you can create a new thing shadow with the same name.

Protocol Support

These methods are supported through both MQTT and a RESTful API over HTTPS. Because MQTT is a pub/sub communication model, AWS IoT implements a set of reserved topics. Things or applications subscribe to these topics before publishing on a request topic in order to implement a request–response behavior. For more information, see Thing Shadows Reserved MQTT Topics (p. 56).

Updating a Thing Shadow

You can update a thing shadow by posting to its RESTful URL or by publishing on the reserved $aws/things/*thingName*/shadow/update topic. Updates affect only the fields specified in the request.

Initial state:

```
 {
    "desired" : {
        "color" : {  "r" :255, "g": 255, "b": 0 }
    }
}
```

An update message is sent:

```
{
    "desired" : {
        "color" : {  "r" : 10 },
            "engine" : "ON"
    }
}
```

Final state:

```
{
    "desired" : {
        "color" : {  "r" : 10, "g" : 255, "b": 0 },
        "engine" : "ON"
    }
}
```

Authorization

Updating a thing shadow requires a policy that allows the principal to perform the `iot:UpdateThingShadow` action. Thing shadows accept two forms of authorization: signature version 4 with IAM credentials or TLS mutual authentication (X.509 client certificates).

The following is an example policy that allows a caller to update a thing shadow:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": "iot:UpdateThingShadow",
        "Resource": ["arn:aws:iot:region:account:thing/thing"]
    }]
}
```

Retrieving a Thing Shadow

You can retrieve a thing shadow by issuing an HTTP GET to the document's RESTful URL or by subscribing and publishing to the reserved $aws/things/*thing-name*/shadows/get topic. This retrieves the entire document plus the delta between the desired and reported states.

Example document:

```
{
    "state" : {
         "desired" : {
             "lights": { "color": "RED" },
             "engine" : "ON"
         },
         "reported" : {
            "lights" : { "color": "GREEN"  },
            "engine" : "ON"
         }
    },
    "metadata" : {
         "desired" : {
             "lights": { "color": { "timestamp" : 123456 },
             "engine" : : { "timestamp" : 123456 }
         },
         "reported" : {
            "lights" : { "color": : { "timestamp" : 789012 }  },
            "engine" : : { "timestamp" : 789012 }
         }
    },
    "version" : 10,
    "timestamp": 123456789
}
```

Request:

```
GET /things/thingName/shadow
```

Response:

```
{
    "state" : {
        "desired" : {
            "lights": { "color": "RED" },
             "engine" : "ON"
         },
         "reported" : {
            "lights" : { "color": "GREEN"  },
            "engine" : "ON"
         },
         "delta" : {
            "lights" : { "color": "RED"  }
         }
    },
    "metadata" : {
        "desired" : {
            "lights": { "color": { "timestamp" : 123456 },
            "engine" : : { "timestamp" : 123456 }
        },
        "reported" : {
            "lights" : { "color": : { "timestamp" : 789012 }  },
            "engine" : : { "timestamp" : 789012 }
         },
         "delta" : {
            "lights" : { "color": { "timestamp" : 123456 }  }
         }
    },
    "version" : 10,
    "timestamp": 123456789
}
```

Authorization

Retrieving a thing shadow requires a policy that allows the calling principal to perform the `iot:GetThingShadow` action. Thing shadows accept two forms of authentication: signature version 4 with IAM credentials or TLS mutual authentication with a client certificate.

The following is an example policy that allows a caller to retrieve a thing shadow:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": "iot:GetThingShadow",
        "Resource": ["arn:aws:iot:region:account:thing/thing"]
    }]
}
```

Optimistic Locking

You can use the state document version to ensure you are updating the most recent version of a thing document. When you supply a version with an update request, the service rejects the request with a HTTP 409 conflict response code if the current version of the state document does not match the version supplied.

For example:

Initial document:

```
 {
    "state" : {
        "desired" : { "colors" : ["RED", "GREEN", "BLUE" ] }
    },
    "version" : 10
}
```

Update: (version doesn't match; request will be rejected)

```
 {
    "state" : { "desired" : { "colors" : [ "BLUE" ] }
    "version" : 9
}
```

Result:

```
409 Conflict
```

Update: (version matches; this request will be accepted)

```
 {
    "state" : { "desired" : { "colors" : [ "BLUE" ] }
    "version" : 10
}
```

Final state

```
{
    "state" : { "desired" : { "colors" : [ "BLUE" ] }
    "version" : 11
}
```

Deleting Data

You can delete data from a thing shadow by using the UPDATE RESTful API or by publishing to the reserved `$aws/things/`*`thingName`*`/shadow/update` topic, setting the fields that are to be deleted to null. Any field with a value of null is removed from the document.

Initial state:

```
 {
    "desired" : {
        "lights": { "color": "RED" },
        "engine" : "ON"
    },
    "reported" : {
        "lights" : { "color": "GREEN"  },
        "engine" : "OFF"
    }
}
```

An update message is sent:

```
 {
    "desired" : null,
    "reported: {
        "engine" : null
    }
}
```

Final state:

```
 {
    "reported" : {
        "lights" : { "color" : "GREEN" }
    }
}
```

Deleting a Thing Shadow

You can delete the entire thing shadow document by using the DELETE RESTful API or by publishing an empty document to the reserved $aws/things/*`{thing-name}/shadows/delete`* topic.

Initial state:

```
 {
    "desired" : {
        "lights": { "color": "RED" },
        "engine" : "ON"
    },
    "reported" : {
        "lights" : { "color": "GREEN"  },
        "engine" : "OFF"
    }
}
```

A delete message is sent:

```
 {
    "state" : null
}
```

Final state:

```
 HTTP 404 - resource not found
```

Delta State

Delta state is a virtual type of state that contains the difference between the desired and reported states. Fields in the desired section that are not in the reported section are included in the delta. Fields that are in the reported section and not in the desired section are not included in the delta. The delta contains metadata, and its values are equal to the desired field's metadata. For example:

```
{
    "state": {
        "desired" : {  "color" : "RED", "state" : "STOP" },
        "reported" : { "color" :"GREEN", "engine" : "ON" },
        "delta" : { "color" : "RED", "state" : "STOP" }
    },
    "metadata" : {
        "desired" : { "color" : {"timestamp" : T1 }, "state" : { "timestamp" :
 T1 },
       "reported" : { "color" : { "timestamp" : T1 }, "engine" : { "timestamp"
 : T2 } },
        "delta"  : { "color" : { "timestamp" : T1 }, "state" : { "timestamp" :
 T1 } }
     },
     "version": 17,
     "timestamp": 123456789
}
```

When nested objects differ, the delta contains the path all the way to the root.

```
{
    "state": {
        "desired" : { "lights" : { "color" : { "r" : 255, "g" : 255, "b" : 255
 } } },
        "reported" : { "lights" : { "color" : { "r" : 255, "g" : 0, "b" : 255
} } },
        "delta" : { "lights" : { "color" : { "g" : 255 } } }
    },
    "version": 18;
    "timestamp": 123456789
}
```

Thing Shadows calculates the delta by iterating through each field in the desired state and comparing it to the reported state.

Arrays are treated like values. If an array in the desired section doesn't match the array in the reported section, then the entire desired array is copied into the delta.

Observing State Changes

When a thing shadow is updated, the AWS IoT publishes two messages. The first message is intended for the requested device and contains only the part of the document that has changed or contains the reason the operation failed. The second message contains the entire document and can be used by other applications or rules to copy the data into another system and process it. Devices and applications can subscribe to either of these topics to be notified when the state of the document has changed.

Here is an example of that flow:

1. Device reports state.
2. The system updates the state document in its persistent data store.
3. The system publishes a *delta* message, which contains only the delta and is targeted at the subscribed devices. Device(s) should subscribe to this topic to receive updates.
4. Thing shadow publishes an *accepted* message, which contains the entire received document, including metadata. Applications should subscribe to this topic to receive updates.

Message Order

There is no guarantee that messages from the AWS IoT service will arrive at the device in any specific order.

Initial state document:

```
{
    "state" : {
        "reported" : { "color" : "blue" }
    },
    "version" : 10,
    "timestamp": 123456777
}
```

Update 1:

```
{
    "state": { "desired" : { "color" : "RED" } },
    "version": 10,
    "timestamp": 123456777
}
```

Update 2:

```
{
    "state": { "desired" : { "color" : "GREEN" } },
    "version": 11 ,
    "timestamp": 123456778
}
```

Final state document:

```
{
    "state": {
        "reported": { "color" : "GREEN" }
    },
    "version": 12,
    "timestamp": 123456779
}
```

This results in two delta messages:

```
{
    "state": { "color" : "RED" }
    "version": 11,
    "timestamp": 123456778
}
```

```
{
    "state": { "color" : "GREEN" },
    "version": 12,
    "timestamp": 123456779
}
```

These messages may be received by the device out of order. Because the state in these messages is cumulative, a device can safely discard any messages that contain a version number older than the one it is tracking. If the device receives the delta for version 12 before version 11, it can safely throw away the version 11 message.

Thing Shadow RESTful Interface

A thing shadow exposes the following URI for updating the state document:

```
/things/thingName/shadow
```

GET

Retrieves the current state document. This includes delta state.

Request:

```
GET /things/thingName/shadow
```

Response:

```
{
    "state": {
        "desired": { "color" : "RED" },
        "delta" : { "color" : "RED}
     },
     "metadata" : {... },
     "version" : 10,
     "timestamp": 123456789
}
```

POST

Updates the state document. The body of the request must be a JSON document and follow the same definition as for an MQTT topic. The `clientToken` parameter is ignored by the HTTP RESTful API.

Request:

```
POST /things/thingName/shadow
```

```
 {
    "state" : { "desired" : { "color" : "RED" } },
    "version: 10
}
```

Response:

```
{
    "state": { ... },
    "metadata" : { ... },
    "version" : 11,
    "timestamp": 123456789
}
```

DELETE

Deletes the thing shadow. The body of the request is empty.

Request:

```
DELETE /things/thingName/shadow
```

Error Messages

Thing Shadows publishes a message on the error topic (over MQTT) when an attempt to change the state document fails. Similarly, they also respond over HTTP RESTful API using the corresponding HTTP error codes. This message is only emitted as a response to a publish on one of the reserved $aws topics. If the client updates the document using the REST API, then it receives the error as part of its response. No MQTT error messages are emitted.

HTTP Error Code	Error Messages
400 (Bad Request)	• Invalid JSON • Missing required node: state • State node must be an object • Desired node must be an object • Reported node must be an object • Invalid version • Invalid clientToken • JSON contains too many levels of nesting; maximum is 6 • State contains an invalid node
401 (Unauthorized)	• Unauthorized
403 (Forbidden)	• Forbidden
404 (Not Found)	• Thing not found
409 (Conflict)	• Version conflict
413 (Payload Too Large)	• JSON document too large
415 (Unsupported Media Type)	• Unsupported documented encoding; supported encoding is UTF-8
500 (Internal Server Error)	• Internal service failures

GetThingState

Request:

```
GET /things/6MviX-S52l_hAh/shadow HTTP/1.1[\r][\n]"
"Host: localhost:8872[\r][\n]"
"Authorization: AWS4-HMAC-SHA256 Credential=AKFakeCredentialWA/2FakeAccount8/us-
east-1/iotdata/aws4_request, SignedHeaders=content-type;host;user-agent;x-amz-
date;x-amz-target, Signature=62344FakeSignature92397468234[\r][\n]"
"X-Amz-Date: 20150928T210240Z[\r][\n]"
"User-Agent: aws-internal/3[\r][\n]"
"X-Amz-Target: IotMoonrakerService.GetThingShadow[\r][\n]"
"Content-Type: application/x-amz-json-1.1[\r][\n]"
"Connection: Keep-Alive[\r][\n]"
"[\r][\n]"
```

Error response:

```
"HTTP/1.1 404 Not Found[\r][\n]"
"content-type: application/json[\r][\n]"
"content-length: 59[\r][\n]"
"date: Mon, 28 Sep 2015 21:02:41 GMT[\r][\n]"
"x-amzn-RequestId: 1cf23eef-ca3a-4606-bc7c-027054ce0c14[\r][\n]"
"connection: Keep-Alive[\r][\n]"
"x-amzn-ErrorType: ResourceNotFoundException:[\r][\n]"
"[\r][\n]"
"{"message":"Thing '725660227939:6MviX-S52l_hAh' not found"}"
```

OK response:

```
"HTTP/1.1 200 OK[\r][\n]"
"content-type: application/json[\r][\n]"
"content-length: 148[\r][\n]"
"date: Mon, 28 Sep 2015 21:02:42 GMT[\r][\n]"
"x-amzn-RequestId: 781a38ce-746c-4e55-b3d1-a65dbc886dd6[\r][\n]"
"connection: Keep-Alive[\r][\n]"
"[\r][\n]"
"{"state":{"reported":{"sequence":"RED,GREEN,RED"}},
  "metadata":{"reported":{"sequence":{"timestamp":1443474162}}},
  "version":1,
  "timestamp":1443474162
}"
[edit][hide]
```

UpdateThingState

Request:

```
"POST /things/6MviX-S52l_hAh/shadow HTTP/1.1[\r][\n]"
"Host: localhost:8872[\r][\n]"
"Authorization: AWS4-HMAC-SHA256 Credential=AFakeCredentialA/2FakeAccount8/us-
east-1/iotdata/aws4_request, SignedHeaders=content-type;host;user-agen
t;x-amz-date;x-amz-target, Signature=76CompletelyFakedSignature9f[\r][\n]"
"X-Amz-Date: 20150928T210241Z[\r][\n]"
"User-Agent: aws-internal/3[\r][\n]"
"X-Amz-Target: IotMoonrakerService.UpdateThingShadow[\r][\n]"
"Content-Type: binary/octet-stream[\r][\n]"
"Transfer-Encoding: chunked[\r][\n]"
```

```
"Connection: Keep-Alive[\r][\n]"
"[\r][\n]"
"4a[\r][\n]"
"{"clientToken":"device","state":{"reported":{"sequence":"RED,GREEN,RED"}}}"
"[\r][\n]"
"0[\r][\n]"
"[\r][\n]"
```

Failure response:

```
"HTTP/1.1 409 Conflict[\r][\n]"
"content-type: application/json[\r][\n]"
"content-length: 30[\r][\n]"
"date: Mon, 28 Sep 2015 21:43:01 GMT[\r][\n]"
"x-amzn-RequestId: 7745824f-fbbe-46fb-897f-c82628e8bff3[\r][\n]"
"connection: Keep-Alive[\r][\n]"
"x-amzn-ErrorType: ConflictException:[\r][\n]"
"[\r][\n]"
"{"message":"Version conflict"}"
```

Response:

```
"HTTP/1.1 200 OK[\r][\n]"
"content-type: application/json[\r][\n]"
"content-length: 171[\r][\n]"
"date: Mon, 28 Sep 2015 21:02:42 GMT[\r][\n]"
"x-amzn-RequestId: b5ed7d8d-a492-457f-abc8-612c5dc8e130[\r][\n]"
"connection: Keep-Alive[\r][\n]"
"[\r][\n]"
"{"state":{"reported":{"sequence":"RED,GREEN,RED"}},
  "metadata":{"reported":{"sequence":{"timestamp":1443474162}}},
  "version":1,
  "timestamp":1443474162,
  "clientToken":"device"}"
```

DeleteThingState

Request:

```
"DELETE /things/6MviX-S52l_hAh/shadow HTTP/1.1[\r][\n]"
"Host: localhost:8872[\r][\n]"
"Authorization: AWS4-HMAC-SHA256 Credential=AFakeCredentialA/2FakeAccount8/us-
east-1/iotdata/aws4_request, SignedHeaders=content-type;host;user-agent;x-amz-
date;x-amz-target, Signature=1643bf8FakeSignature88db0[\r][\n]"
"X-Amz-Date: 20150928T210249Z[\r][\n]"
"User-Agent: aws-internal/3[\r][\n]"
"X-Amz-Target: IotMoonrakerService.DeleteThingShadow[\r][\n]"
"Content-Type: application/x-amz-json-1.1[\r][\n]"
"Connection: Keep-Alive[\r][\n]"
"[\r][\n]"
```

Response:

```
"HTTP/1.1 200 OK[\r][\n]"
"content-type: application/json[\r][\n]"
"content-length: 36[\r][\n]"
"date: Mon, 28 Sep 2015 21:02:49 GMT[\r][\n]"
"x-amzn-RequestId: 8c2a61b2-c37f-4e07-b3ea-481267bae871[\r][\n]"
"connection: Keep-Alive[\r][\n]"
"[\r][\n]"
"{"version":3,"timestamp":1443474169}"
```

Thing Shadow MQTT Pub/Sub Messages

The following are the pub/sub messages used with MQTT for interacting with thing shadows.

Messages

/update/accepted

Messages published to $aws/things/*thingName*/shadow/update/accepted have the following form:

```
{
    "state": {
        "reported": {"sequence":"RED,GREEN,RED"}
    },
    "metadata": {
        "reported": {"sequence": {"timestamp": 1443474162}}
    },
    "version": 1,
    "timestamp": 1443474162,
    "clientToken": "device"
}
```

/update/rejected

Messages published to $aws/things/*thingName*/shadow/update/rejected have the following form:

```
{
   "code": 409,
   "message": "Version conflict",
   "clientToken": "app"
}
```

/update/delta

Messages published to $aws/things/*thingName*/shadow/update/delta have the following form:

```
{
   "version": 2,
   "state": {"sequence":"BLUE,YELLOW,GREEN"},
   "metadata": {"sequence":{"timestamp":1443474163}},
    "timestamp": 1443474163,
    "clientToken": "app"
}
```

/get/accepted

Messages published to $aws/things/*thingName*/shadow/get/accepted have the following form:

```
{
    "state": {"reported":{"sequence":"RED,GREEN,RED"}},
    "metadata": {"reported":{"sequence":{"timestamp":1443474183}}},
    "version": 1,
    "timestamp": 1443474183,
    "clientToken": "device"
}
```

/get/rejected

Messages published to $aws/things/*thingName*/shadow/get/rejected have the following form:

```
{
    "code": 404,
    "message": "Thing '725660227939:czvDd-NjGW_IYZ' not found",
    "clientToken": "device"
}
```

/delete/accepted

Messages published to $aws/things/*thingName*/shadow/delete/accepted have the following form:

```
{
    "version": 3,
    "timestamp": 1443474169,
    "clientToken": "device"
}
```

/delete/rejected

Messages published to $aws/things/*thingName*/shadow/delete/rejected have the following form:

```
{
     "code": 404,
     "message": "Thing '725660227939:sPfhn-JPs9_A5C' not found",
     "clientToken": "device"
}
```

Thing Shadows Reserved MQTT Topics

Thing shadows uses reserved MQTT topics to allow applications and things to get, update, or delete thing state information. The names of these topics start with "$aws/things/*thingName*/shadow."

Publishing and subscribing on thing shadow topics requires topic-based authorization.

Topics

update

A thing publishes on this topic to update the JSON state document:

`$aws/things/thingName/shadow/update`

You can specify a version attribute in the JSON document to specify which version of the JSON document you are attempting to update. When a version attribute is present in an MQTT message, AWS IoT processes the update only if the message version matches the latest version of the state document in AWS IoT. The following example shows a message with a version attribute:

```
{
   "state" : {
      "desired": {
         "color": "RED",
         "temperature": 55
      },
      "reported": {
         "color": "GREEN",
         "temperature": 35
      }
   },
   "version": 42,
   "clientToken": "UniqueForTheClient"
}
```

Example Policy

The following is an example policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Publish"],
       "Resource": ["arn:aws:iot:region:account:topic/$aws/things/thingName/shad
ow/update"]
     }]
}
```

update/accepted

AWS IoT publishes to this topic when it accepts a change:

$aws/things/*thingName*/update/accepted

The message body contains the updated state. For example, if the following JSON document was published on $aws/things/*thingName*/update:

```
{
   "state": { "desired": { "color" : "RED" } }
}
```

AWS IoT publishes the following JSON on $aws/things/*thingName*/update/accepted:

```
{
   "state": { "desired": { "color" : "RED" } }
   "version": 43,
   "metadata": { "desired" : { "color" : { "timestamp" : 100 } } }
}
```

Example Policy

The following is an example policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Subscribe"],
       "Resource": ["arn:aws:iot:region:account:topic/$aws/things/thingName/shad
ow/update/accepted"]
    }]
}
```

update/rejected

AWS IoT publishes on this topic when it rejects a change:

$aws/things/*thingName*/shadow/update/rejected

For example:

```
{
   "code": 400,
   "message": "invalid json",
   "timestamp": 100123,
   "clientToken": "1234",
   "version": 12
}
```

The following parameters are used in every rejected message:

- `code` — An HTTP response code that indicates the type of error.
- `message` — A nonstandardized text message that provides additional human-readable error information.
- `timestamp` — The date and time the message was generated by AWS IoT.
- `clientToken` — Present only if a client token was used in the preceding publish to update.
- `version` — Enables devices to synchronize on the current shadow document version.

Example Policy

The following is an example policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Subscribe"],
       "Resource": ["arn:aws:iot:region:account:topic/$aws/things/thingName/shad
ow/update/rejected"]
    }]
}
```

update/delta

AWS IoT publishes on this topic when it accepts a change and the JSON state document contains different values for desired and reported states.

$aws/things/*thingName*/shadow/update/delta

The following rules determine when messages are published to update/delta and what is included in the messages:

- Messages published on the update/delta topic contain only the desired attributes that differ between the reported and desired sections. The messages contains all of these attributes, regardless of whether these attributes were contained in the current update message or were already stored in AWS IoT. Attributes that do not differ between reported and desired are not included in messages published on update/delta.
- If an attribute is in the reported section but has no equivalent in the desired section, it is not included in messages published on update/delta.
- If an attribute is in the desired section but has no equivalent in the reported section, it is not included in messages published on update/delta.
- If an attribute is deleted from the reported section but still exists in the desired section, it is included in messages published on update/delta.

Note
The state and reported sections are attributes contained in the JSON document.

The following examples illustrate a scenario in which a device publishes its state and then an app updates the device state:

AWS IoT contains to following JSON document for a thing:

```
{
   "state": { "color" : "RED" }
   "version": 10,
   "timestamp": 1234567,
   "metadata": { "color" : { "timestamp" : 100 } }
}
```

A thing publishes the following message on update:

```
{
   "state": { "reported": { "color" : "RED" } }
}
```

A controller app attempts to update the thing state:

```
{
   "state": { "desired": { "color" : "GREEN" } }
}
```

AWS IoT publishes the following on update/delta:

```
{
   "state": { "color" : "GREEN" }
   "version": 12,
   "timestamp": 2345679
   "metadata": { "color" : { "timestamp" : 2345678 } }
}
```

Example Policy

The following is an example policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Subscribe"],
       "Resource": ["arn:aws:iot:region:account:topic/$aws/things/thingName/shad
ow/update/delta"]
    }]
}
```

delete

A thing publishes an empty document on this topic to delete the JSON state document:

`$aws/things/thingName/shadow/delete`

Example Policy

The following is an example policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Subscribe"]
       "Resource": ["arn:aws:iot:region:account:topic/$aws/things/thingName/shad
ow/delete"]
    }]
}
```

delete/accepted

AWS IoT publishes a message on this topic when deleting a thing shadow:

$aws/things/*thingName*/shadow/delete/accepted

Example Policy

The following is an example policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Subscribe"]
       "Resource": ["arn:aws:iot:region:account:topic/$aws/things/thingName/shad
ow/delete/accepted"]
    }]
}
```

delete/rejected

AWS IoT publishes a message on this topic when a thing shadow cannot be deleted:

$aws/things/*thingName*/shadow/delete/rejected

Example Policy

The following is an example policy:

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": ["iot:Subscribe"]
       "Resource": ["arn:aws:iot:region:account:topic/$aws/things/thingName/shad
ow/delete/rejected"]
```

```
    }]
}
```

AWS IoT SDKs

Contents

AWS IoT provides a number of ways to interact with the service platform, using the MQTT and HTTP RESTful protocols, and across different language frameworks. Depending on your application use cases, you may find one or more SDKs to use.

- **AWS SDKs for configuration**: All configuration APIs are available as part of the standard AWS SDKs in all supported AWS SDK languages, including SWIFT for iOS9, under the package namespace "iot." There are functions to create and manage identities and authorization (including certificates and policies), rules and actions in the rules engine, endpoints, logging roles and levels, as well as the names and attributes for things in the Thing Registry.
- **AWS SDKs to publish messages and work with thing shadows**: AWS IoT provides support for:
  - Applications using IAM users/roles (either directly or through Amazon Cognito).
  - The HTTP protocol to publish messages directly to the message broker using HTTP POST.
  - The ability to get, update, and delete states for thing shadows.

  These APIs are available in all languages supported by the AWS SDKs, including SWIFT for iOS9, under the package namespace "iot-data."
- **Thing SDKs**: AWS IoT provides support for devices and applications that use certificate-based authentication and the MQTT protocol (typically, memory-constrained devices) to perform message publish and subscribe to the message broker. The C SDK can be compiled on Linux distributions using OpenSSL or mbedTLS libraries and the included MQTT library. Customers can also port this SDK to additional embedded platforms and incorporate custom TLS and MQTT libraries, if needed. A special version of the C SDK is provided for Arduino Yún. The Thing SDK is available for JavaScript in the Node.js package ecosystem as well as in SWIFT for iOS9.

Device SDK for AWS IoT

The Thing SDK is a C language SDK developed for constrained devices or system on a chip (SoC) with 256 KB or more of available memory. The Thing SDK simplifies the process of connecting embedded devices to the AWS IoT platform by implementing the security requirements to connect to the broker of the AWS IoT service and providing basic pub/sub functionality as well as access to Thing Shadows over MQTT. This eliminates the need to implement support for an HTTP RESTful API. The Thing SDK was designed and tested to work with embedded Linux and a variety of industry-standard, real-time operating systems as well as third-party TLS 1.2 and cryptographic library implementations to support the high security standards of the broker of the AWS IoT platform.

Thing SDK Feature Set

The Thing SDKs provide this functionality across platforms:

- Configuration of security credentials and artifacts for TLS 1.2 mutual authentication communication with the AWS IoT message broker in the supported regions.
- Establishes and manages connection with the AWS IoT message broker (over MQTT) including timeout configuration.
- Provides a wrapper for common MQTT client implementations to publish data and subscribe to topics over MQTT.
- Provides support for updating, retrieving, and deleting thing shadows, including support for the versioning of thing shadows.

Platform Support

The device SDK is available for starter kits for rapid prototyping and for major semiconductor manufacturers' evaluation kits for product development. The currently supported platforms are:

- **Arduino Yún**

  The Arduino Yún features a powerful communication processor that allows you to use TLS 1.2 and client certificates to connect the Arduino Yún board to AWS IoT. The SDK includes installation instructions and scripts that prepare your Arduino Yún board for connection to AWS IoT. You can now do even more with your Arduino sketch by having access to the cloud features of AWS through AWS IoT.

  For more information, see:

  AWS IoT API Reference

  AWS IoT Quickstart

  AWS IoT SDK for Arduino Readme

  Download the AWS IoT SDK for Arduino
- **C SDK for embedded platforms (Linux, real-time OS)**

  The C SDK was developed following the C99 language standard. The SDK provides sample applications and getting started instructions [LINK] to improve the onboarding experience and quickly connect your device to AWS IoT. The C SDK also includes a porting guide that helps you modify the SDK's wrapper functions to support your platform-specific data types, OS timers, and embedded TLS implementation, so that your application code can remain unchanged when moving from a rapid prototyping environment in Linux to a more constrained and cost-efficient microcontroller.

  For more information, see:

  AWS IoT API Reference

  AWS IoT Quickstart

  AWS IoT SDK for C
- **Node.js JavaScript runtime package**

The Node.js SDK was developed for rapid prototyping on more powerful embedded platforms that support the Node.js package ecosystem. You can use the NPM packager manager to install the AWS IoT package for Node.js. The example application and instructions will demonstrate which additional packages are needed to run your Node.js application and connect it to AWS IoT quickly.

For more information, see:

AWS IoT API Reference

AWS IoT Quickstart

AWS IoT SDK for JavaScript

Node.js SDK for AWS IoT

This section discusses sample applications available for the Node.ja SDK for AWS IoT. All samples require a root CA certificate, which you can download from Symantec.

On the Licensing and Use of Root Certificates web page, scroll down to **Root 3** and choose the Download Root Now link. Save the file on your computer.

These samples also require an AWS IoT-generated certificate and private key. For more information, see Provision a Certificate.

BeagleBone Sample

Use the Node.js SDK for &IoT; to create a sample application that could be distributed with the SDK.

To get started with BeagleBone Black:

1. Plug in your Beagle with a USB cable.
2. Install the required drivers.
3. Update to the latest system image.
4. Browse to the web server on Beagle.

Configuring Cloud9 IDE

Cloud9 IDE is the web-based programming platform on BeagleBone Black.

To enter the IDE,,in the browser, type <*`IP address of the board`*> : 3000 (for example, 192.168.7.2:3000). The left side of the screen is a workspace. You can access your folders and files from the workspace window. Everything is organized in a hierarchy.

In Cloud9 IDE, choose **File**, choose **Upload Local Files**, and then upload the BeagleboneBlack folder from the Node.js SDK to the IDE workspace.

Use `npm` to import dependencies. Use the `node -v command` to see if the Node.js version is greater than 0.10. Connect to the board by using SSH (or in the IDE, there is a bash-beaglebone window).

To go to the Cloud9 folder, type:

```
cd /var/lib/cloud9
```

In the terminal, type:

```
npm install mqtt --save
```

This will create the node-module folder.

Copy the root CA certificate, the device certificate, and your private key into the prodCerts folder. By default, the prodCerts folder has three empty files. The sample code is written with the assumption these are the names of your certificates and private key. You can rename your files to match these empty files or modify the sample code to use your file names.

Run the Samples

In the example folder, there are two applications: the LED application (digital) and the potentiometer application (analog). Each application has one pub and one sub. Run the application on the IDE or connect to the BeagleBone by using SSH. Type `node ..js` to run the application in the terminal .

In the LED application, the LED is connected to pin P9_11 on the board. You can publish a HIGH or LOW value in any device. If the BeagleBone Black board subscribes it, the LED on the board will be on or off according to the value you publish.

There are three wires in the potentiometer application. The first wire connects to VDD_3V3, the second wire connects to pin P9_9, and the third wire connects to ground. The BeagleBone Black can publish the value of the potentiometer continuously. You can subscribe it in any device to see the value of the potentiometer.

Intel Edison Sample

To get started with the Intel Edison, see Getting Started with Intel Edison Technology.

There are two ways to run Node.js files: connect by using SSH or use the Intel XDK IDE.

Connect to Intel Edison by Using SSH

Copy the project folder into the board, and then run:

```
npm install mqtt
```

Copy the certificates into the prodCerts folder. Use this command to run the XXX.js in the example:

```
node XXX.js
```

Use the Intel XDK IoT Edition IDE

Run the Intel Edison integrated installer, and choose the Intel XDK IoT Edition IDE. The IDE enables you to program in JavaScript and Node.js.

Set up the IDE and create a project. For more information, see Using the Intel XDK IoT Edition.

Copy the src folder, prodCerts folder, and example folder into your project. Copy the root CA certificate, your device certificate, and your private key into the prodCerts folder. By default, the prodCerts folder has three empty files. The sample code is written with the assumption these are the names of your certificates and private key. You can rename your files to match these empty files or modify the sample code to use your file names.

The src folder contains aws_iot.js, which is the Node.js SDK for AWS IoT. Add this to your application.

In the example folder, include the certificates in the client_params, because the default path cannot be found in the IDE. For example:

- key: *'/node_app_slot/prodCerts/device_key.pem'*
- certificate: *'/node_app_slot/prodCerts/device_identity.pem'*
- CA: *'/node_app_slot/prodCerts/VeriSign-Class 3-Public-Primary-Certification-Authority-G5.pem'*

When you upload your project into the board, the /node_app_slot folder is created for you.

Create a file named package.json. In package.json, add the following properties: "name", "version", "main", "dependencies". The "main" value is the application to run. The "dependencies" value is the node_modules your project uses. For example:

```
{
    "name": "aws_iot",
    "version": "1.0.0",
    "main": "example/Helloworld.js",
    "license": "ISC",
    "dependencies": {
        "keypress": "^0.2.1",
        "mocha": "^2.2.5",
        "mqtt": "^1.3.5"
    }
}
```

Raspberry Pi Sample

To get started with Raspberry Pi, see Getting Started with Raspberry Pi.

Install Node.js

Sign in to your Raspberry Pi and run this command to add the package repository:

```
curl -sLS https://apt.adafruit.com/add | sudo bash
```

Use `apt-get` to install the latest version of Node.js.:

```
sudo apt-get install node
```

Run the Samples

Use SFTP to upload the project folder to the board.

In the root folder of the project, run this command to install MQTT:

```
npm install mqtt
```

Copy the root CA certificate, the device certificate, and your private key into the prodCerts folder. By default, the prodCerts folder contains three empty files. The sample code is written with the assumption these are the names of your certificates and private key. You can rename your files to match these empty files or modify the sample code to use your file names.

From the project folder, run the JS files.

Troubleshooting AWS IoT

Diagnosing Connectivity Issues

Authentication

How do my devices authenticate AWS IoT endpoints?

Add the AWS IoT certification authority (CA) certificate to your client's trust store. You can download the CA certificate here.

How can I validate a correctly configured certificate?

Use the OpenSSL `s_client` command to test a connection to the AWS IoT endpoint:

```
openssl s_client -connect iot.us-east-1.amazonaws.com:443 -CAfile CA.pem
-cert cert.pem -key privateKey.pem
```

Authorization

I received a `PUBNACK` or `SUBNACK` response from the broker. What do I do?

Make sure there is a policy attached to the certificate you are using to call AWS IoT. All publish/subscribe operations are denied by default.

Setting up CloudWatch Logs

As messages from your devices pass through the message broker and the rules engine, AWS IoT sends progress events about each message. You can opt in to view these events in CloudWatch Logs.

Note
You should understand the access permissions to CloudWatch Logs within your AWS account before enabling AWS IoT logging. Users with access to CloudWatch Logs will be able to see debugging information from your devices

For more information about CloudWatch Logs, see CloudWatch Logs .

Note
Before you enable AWS IoT logging, be sure you understand the access permissions to CloudWatch Logs in your AWS account . Users with access to CloudWatch Logs will be able to see debugging information from your devices.

Configuring an IAM Role for Logging

Create an IAM Role for Logging

Use the IAM console to create a logging role. The following policy documents provide the role policy and trust policy that allow AWS IoT to submit logs to CloudWatch on your behalf.

Role policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents",
                "logs:PutMetricFilter",
                "logs:PutRetentionPolicy",
             ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Trust policy:

```
{
 "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": "iot.amazonaws.com",
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Register the Logging Role with AWS IoT

Use the AWS IoT console or the following CLI command to register the logging role with AWS IoT.

```
aws iot set-logging-options --logging-options-payload
roleArn=”arn:aws:iam::<your-aws-account-num>:role/IoTLoggingRole”,logLevel=”INFO”
```

The log level can be one of DEBUG, INFO, ERROR, or DISABLED:

- DEBUG provides the most detailed set of information on what happens in AWS IoT
- INFO provides a summarized view of most actions. This is sufficient for most users
- ERROR provides only error error cases

- DISABLED removes logging altogether but keeps your logging role intact

View the CloudWatch Logs

This section lists the logging events and error codes sent by AWS IoT.

Identity and Security

Operation/Event Name	Description
Authentication Success	Successfully authenticated a certificate.
Authentication Failure	Failed to authenticate a certificate.

Identity and Security Error Codes

Error Code	Error Description
401	Unauthorized

Message Broker

Operation/Event Name	Description
MQTT Publish	MQTT Publish received.
MQTT Subscribe	MQTT Subscribe received.
MQTT Connect	MQTT Connect received.
MQTT Disconnect	MQTT Disconnect received.
HTTP/1.1 POST	MHTTP/1.1 POST received.
HTTP/1.1 GET	HTTP/1.1 GET received.
Throttling Alarm	A device is approaching or exceeding its throttling limit.
HTTP/1.1 Unsupported Method	Used when message contains syntax error or action forbidden (HTTP PUT/DELETE/...).
Malformed HTTP Messsage	The connection was terminated because of a malformed HTTP message.
Malformed MQTT Message	The connection was terminated because of a malformed MQTT message.
Connection Limit Exceeded	The connection was refused because the connection limit was exceeded.
Authorization Failed	This client attempted to publish to or subscribe on a topic for which it has no authorization.
Package Exceeds Maximum Payload Size	This client attempted to publish a payload that exceeds the message broker's upper limit.

Message Broker Error Codes

Error Code	Error Description
400	Bad Request
401	Unauthorized
403	Forbidden
503	Service Unavailable

Rues Engine Events

Operation/Event Name	Description
MessageReceived	Received request for a topic.
DynamoActionSuccess	Successfully put DynamnoDB record.
DynamoActionFailure	Failed to put DynamoDB record.
KinesisActionSuccess	Successfully published Amazon Kinesis message.
KinesisActionFailure	Failed to publish Amazon Kinesis message.
LambdaActionSuccess	Successfully invoked Lambda function.
LambdaActionFailure	Failed to invoke Lambda function.
RepublishActionSuccess	Successfully republished message.
MessageReceived	Received request for a topic.
RepublishActionFailure	Failed to republish message.
S3ActionSuccess	Successfully put S3 object.
S3ActionFailure	Failed to put S3 object.
SNSActionSuccess	Successfully published to SNS topic.
SNSActionFailure	Failed to publish to SNS topic.
SQSActionSuccess	Successfully sent message to SQS.
SQSActionFailure	Failed to send message to SQS.

Thing Shadow Events

Operation/Event Name	Description
UpdateThingState	A thing's state is updated over HTTP or MQTT.
DeleteThing	A thing is deleted.

Thing Shadow Error Codes

Error Code	Error Description
400	Bad request
401	Unauthorized
403	Forbidden
404	Not found
409	Conflict
413	Request too large
422	Failed to process request
429	Too many requests
500	Internal error
503	Service unavailable